

Computer Organization

Introduction to Verilog

Dongup Kwon (dongup@snu.ac.kr)

High Performance Computer Systems (HPCS) Lab.

March 16, 2017

Project

- In this project, you will design and implement an **in-order pipelined CPU** using Verilog.
- Tools
 - Hardware description language (HDL): Verilog
 - Design & simulation tool: Xilinx Vivado Design Suite
- Tentative schedule

3/16	Introduction to Verilog and ALU
3/23	Synchronous Circuit and Sequence Detector
3/30	Register-Transfer Level (RTL)
4/6	Single-cycle CPU
4/13	Multi-cycle CPU
4/20	Pipelined CPU
5/4	Cache
5/11	DMA

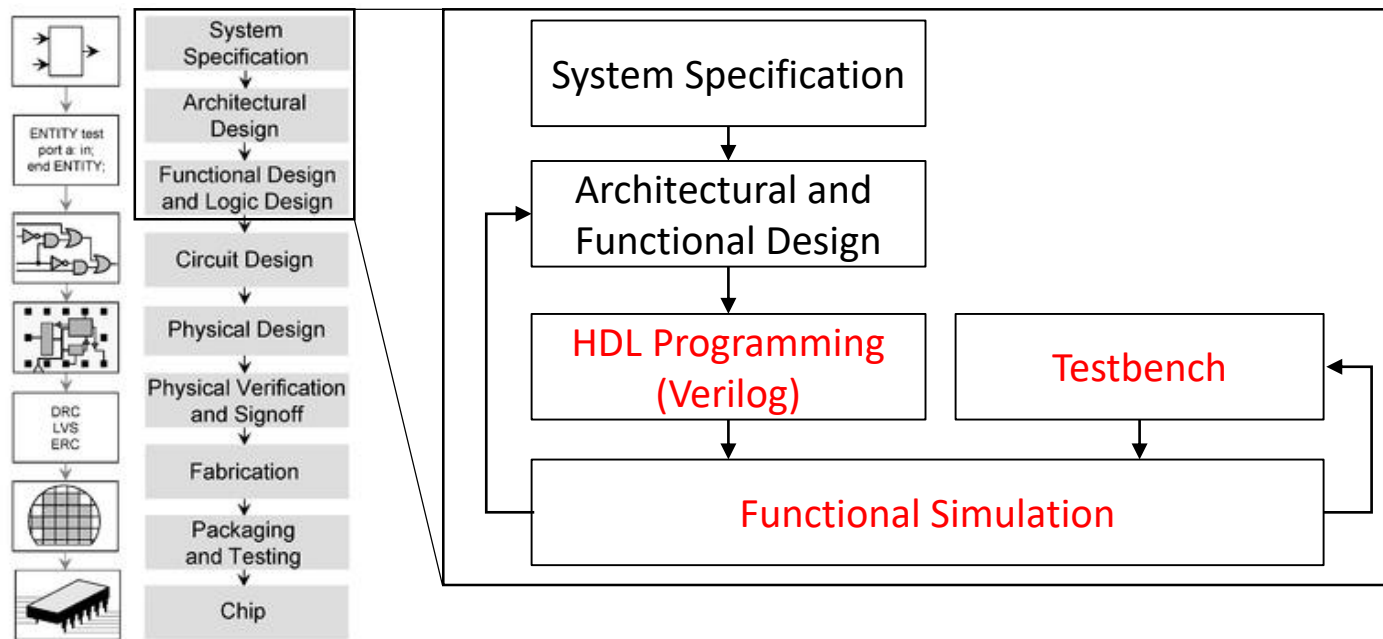
Overview

- In this project, you will learn **how to describe digital circuits in Verilog** and **how to use design tools** (*e.g., Vivado Design Suite*).
- All the assignments are related; that is, you need to complete all previous steps to do next assignment.
- Your assignment for this week is implementing a simple 16-bit ALU. This also will be used in the CPU.
- **Goals**
 - Understand how to describe digital circuits in Verilog
 - Implement an ALU (i.e., a basic functional unit in CPUs)

Hardware Description Language (HDL)

HDL and Verilog

- **HDL:** a formal notation designed to describe *electronic circuits*
- **Verilog:** most widely used and well-supported HDL



Design Steps [1]

Verilog Programming

How to describe 2-input AND gate

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Source code

- Timescale
- Module
 - Ports
 - Variables
 - Behavior models (body)
 - Assignments
 - if/case/loop statements
 - Timing controls
 - Structured procedures

Code structure

Timescale

- ``timescale time_unit / time_precision`
 - Example: ``timescale 1 ns / 100 ps`
- *time_unit*: the unit of measurement
 - Example: `#100 = 100 * 1ns delay`
- *time_precision*: the unit of simulation
- *time_precision* cannot specify a longer unit of time than *time_unit*.

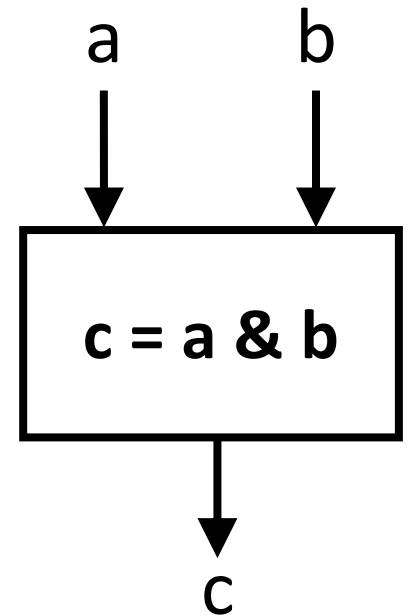
Module

- A unit of hierarchical hardware structure
 - High-level modules create the instances of low-level modules and communicate with them through ports.
 - A *top-level module* is the highest-level module.
- Interfaces + behavior models

```
module add (a, b, c);  
    input a;  
    input b;  
    output c;  
    assign c = a & b;  
endmodule
```

Interfaces (ports)

Behavioral models



Module - Instantiation

- *Module instantiation* allows one module to create a copy of another module into itself.
- Possible connections : wire to wire, wire to reg, and reg to wire.

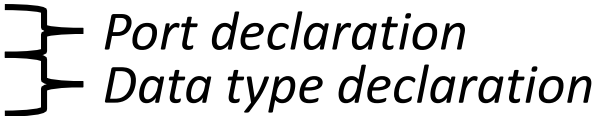
```
module dummy (in, out);  
    input in;  
    output out;  
    reg out;  
endmodule
```

```
module user ();  
    reg x  
    wire y;  
    dummy inst1 (.in(x), .out(y));  
endmodule
```

Port

- A means of interconnecting modules
 - Optional, but usually necessary
- Port declaration: *input*, *output*, *inout* (bidirectional)
- Data type declaration: *wire* (default), *reg*

```
module always_one (x);  
    output x;  
    reg x;  
    initial x <= 1;  
endmodule
```



Port declaration

Data type declaration

Data Type

- Data types is designed to represent the *data storage* and *transmission elements*.
- Values
 - 0: a logic zero
 - 1: a logic one
 - x: an unknown logic value
 - z: a high-impedance state
- *net* and *variable* data-type groups
 - *net*: a representation of physical connections (e.g., *wire*)
 - *variable*: an abstraction of data storage elements (e.g., *reg*)
- We will use only *wire* and *reg* data types
- Declaration
 - `wire [0:31] addr; // Big endian`
 - `reg [31:0] eax; // Little endian (Recommended)`

Data Type – net

- It defines a physical connection with a continuous assignment.
 - *wire* data type is the most representative. We use only this.
- Example
 - wire c;**
 - assign c = a & b;**
 - Then **c** is “physically connected” to the result of the “AND gate”.
 - *wire* data type usually describes a combinational circuit.

Data Type – variable

- It is an abstraction of “storage elements” which can be read and written.
 - *reg* data type is the most representative. We use only this.
- It is mainly used in a sequential circuit.

Syntax

- Case sensitive
- Comments : `//` or `/* */` (Similar to C language)
- Identifier : `[a-zA-Z_]([a-zA-Z0-9_$])+` (Up to 1024 chars)
- Integer number : `<size>'<radix><value>`
 - `<size>` : # of bits
 - `<radix>` : h(hex), d(decimal), o(octal), b(binary)
 - `<value>` : Depend on the radix
 - X : “Unknown”, Z : “High Impedance”, _ : (Ignored)

Operators

- Arithmetic
 - Binary : +, -, *, /, %
 - Unary : +, -
- Relational
 - Binary : <, >, <=, >=
- Equality
 - Binary : ==, !=
 - (comparing include x & z)
 - Binary : ==, !=
 - (returns x if a operand has x or z)
- Logical
 - Unary : !
 - Binary : &&, ||
- Bit-wise
 - Unary : ~
 - Binary : &, |, ^, ~^
- Reduction
 - Unary : &, ~&, |, ~|, ^, ~^
- Shift
 - Binary : <<, >>
- Concatenation
 - {a, b, c, ...}
- Replication
 - {n, {m}}
- Conditional
 - a?b:c

Behavioral Model

- Continuous assignments
- Procedural assignments
- if statement
- Case statement
- Loop statement
- Timing controls
- Structured procedures

Continuous Assignments

- It drives values onto *net* or defines a connection for *net*.
 - **wire** c;
 - **assign** c = a & b; // The type of c should be the net.
- With a declaration of *wire*
 - **wire** c = a & b;

Continuous Assignments

- With timing controls
 - **assign** #10 c = a & b; // c is updated after 10 delay unit since a or b is updated.
- Unless it has a delay modeling, we can regard it as an aliasing.
 - **assign** y = x; // y is an alias of x.
 - **assign** c = a & b; // c is an alias of (anonymous net data) 'a & b'

Procedural Assignments

- It puts values in *variable*, and *variable* holds the value until the next procedural assignment to that *variable*.
- It occurs within *procedures* such as **always** and **initial**.
 - **reg** x;
 - **always** x = 1; // The data type of x should be *reg*.
- With timing controls
 - #10 x = c; // After 10 delay unit, store the value of c to x.
 - #10 x = #20 c; // After 10 delay unit, read the value of c. Then, after additional 20 delay unit, store it to x.
 - @e x = c; // Store the value of c to x when event e is occurred.

Non-blocking & Blocking Assignment

- The two mechanism of *procedural assignments*
- It is really important but confusing, so you have to understand it very well.

Non-blocking (symbol: ' \leq ')

- It **doesn't specify** the execution order of statements.

```
module example ();  
    reg a, b;  
    initial begin  
        a <= 0;  
        b <= 1;  
    end  
endmodule
```

≡

```
module example ();  
    reg a, b;  
    initial begin  
        b <= 1;  
        a <= 0;  
    end  
endmodule
```

Blocking (symbol: '=')

- It **defines** the execution order of statements.
- Similar to the non-blocking, it stores the value.
- **However, it is different from the non-blocking in respecting the order of assignments.**

```
module example ();  
  reg a, b;  
  initial begin  
    a = 0;  
    b = 1;  
  end  
endmodule
```

≡

```
module example ();  
  reg a, b;  
  initial a = 0;  
  initial b = 1;  
endmodule
```

≠

```
module example ();  
  reg a, b;  
  initial b = 0;  
  initial a = 1;  
endmodule
```

Blocking vs. Non-blocking

```
module example ();  
  reg a, b, c;  
  initial begin  
    a <= 0;  
    b <= 1;  
    initial begin  
      #50 b << a;  
      c << b;  
    end  
  end  
endmodule
```

<< is blocking (=)

Cycle	a	b	c
0	0	1	x
50	0	0	0
100	0	0	0

<< is non-blocking (<=)

Cycle	a	b	c
0	0	1	x
50	0	0	1
100	0	0	1

Mixed Blocking & Non-blocking

- Don't even think about it!
- There is no syntactic error, but it is hard to predict the consequence, especially when they are used with “if” or “for”.
- In this lab, we recommend you to use the *blocking assignments*.
 - The *blocking assignment* is mainly used to describe combinational circuits.
- We will introduce the usage of *non-blocking assignment* in the next project assignment.

If statement

- It executes a statement conditionally.

```
module example ();  
    // ...  
    initial begin  
        if (a == 5)  
            b <= 15;  
        else  
            b <= 25;  
    end  
endmodule
```

≡

```
module example ();  
    // ...  
    wire c = a == 5 ? 15 : 25  
    initial begin  
        b <= c;  
    end  
endmodule
```

Case statement

- It executes a statement conditionally.

```
module example ();  
    // ...  
    initial begin  
        if (a == 5)  
            b <= 15;  
        else  
            b <= 25;  
    end  
endmodule
```

≡

```
module example ();  
    // ...  
    initial begin  
        case (a)  
            5 : b <= 15;  
            default : b <= 25;  
        endcase  
    end  
endmodule
```

Loop statement - repeat

```
module example ();  
    // ...  
    initial begin  
        repeat (4) c = c + 1;  
    end  
endmodule
```

≡

```
module example ();  
    // ...  
    initial begin  
        c = c + 1;  
        c = c + 1;  
        c = c + 1;  
        c = c + 1;  
    end  
endmodule
```

Loop statement - forever

- Same as repeat (∞)

```
module example ();  
    // ...  
    initial begin  
        forever #5 c = c + 1;  
    end  
endmodule
```

≡

```
module example ();  
    // ...  
    initial begin  
        #5 c = c + 1;  
        #5 c = c + 1;  
        #5 c = c + 1;  
        // infinitely repeated  
    end  
endmodule
```

Loop statement – while, for

- It describes a repetition task, but sometime it cannot be implemented as a real circuit.
- The typical usage is an array initialization.

```
module example ();  
    initial begin  
        while (c < 10)  
            c = c + 1;  
    end  
endmodule
```

```
module example ();  
    initial begin  
        for (c=0; c<10; c++)  
            // ...  
    end  
endmodule
```

Timing Controls - Delay & Event

- *Delay* and *event* are two methods for specifying when the procedural assignments occurs.

Delay

- Its meaning can be different along the context.
 - We already explains two cases.
- It is used for describing the timing of real hardware, but not for controlling behaviors of Verilog code.
- So, you must not use “delay”, except for a clock signal and a testbench.

Event

- We can implement an event-driven code.
 - e.g., Modeling an edge-triggered flip-flop.

```
module add (a, b, c);  
    reg on;  
    event e;           } Event declaration  
    initial begin  
        on <= 0;  
        #50 -> e;      } Event trigger  
    end  
    initial @e on <= 1; } Event-triggered assignment  
endmodule
```

Event

- The change of data is an event by itself.
- We can make an or-trigger.
 - e.g., (e1 or e2)
- posedge and negedge means 0 to 1 and 1 to 0 respectively.

```
module example ();  
    reg clk;  
    reg [31:0] cnt;  
    always begin  
        @(posedge clk) cnt = cnt + 1;  
    end  
endmodule
```

Event

- We can specify the same event to all statements in a begin ... end block.

```
module always_one ();  
    // ...  
    always  
    begin  
        @(posedge clk) x <= 1;  
        @(posedge clk) y <= 0;  
    end  
endmodule
```

≡

```
module always_one ();  
    // ...  
    always @(posedge clk)  
    begin  
        x <= 1;  
        y <= 0;  
    end  
endmodule
```

Event

- We can represent a complex combinational circuit with an event.

```
module example ();  
    wire a, b, c;  
    assign c = ~(a & b);  
endmodule
```

≡

```
module example ();  
    wire a, b;  
    reg c;  
    always @(a or b) begin  
        c = ~(a & b);  
    end  
endmodule
```

Structured Procedures

- **always, initial**
- They are necessary for designing sequential circuit.

Structured Procedures - initial

- It is executed once at the beginning.

```
module example ();  
    reg clk, x;  
    initial clk <= 0;  
    initial x <= 1;  
endmodule
```

≡

```
module example ();  
    reg clk, x;  
    initial begin  
        clk <= 0;  
        x <= 1  
    end  
endmodule
```

- It initializes reg clk and x.
- It can involve multiple statements with begin ... end.

Structured Procedures - initial

- The delays are accumulated in a begin ... end block.

```
module example ();  
    reg clk;  
    initial #50 clk <= 0;  
    initial #100 clk <= 1;  
    initial #150 clk <= 0;  
endmodule
```

≡

```
module example ();  
    reg clk;  
    initial begin  
        #50 clk <= 0;  
        #50 clk <= 1;  
        #50 clk <= 0;  
    end  
endmodule
```

Structured Procedures - initial

- You can write delay statements separately.

```
module example ();  
    reg clk;  
    initial begin  
        #50 clk <= 0;  
        #50 clk <= 1;  
    end  
endmodule
```

≡

```
module example ();  
    reg clk;  
    initial begin  
        #50;  
        clk <= 0;  
        #50;  
        clk <= 1;  
    end  
endmodule
```


Structured Procedures - always

- **always** is equal to the infinite repetition of a initial block.

```
module example ();  
    reg clk;  
    initial clk <= 0;  
    initial begin  
        #50 clk <= ~clk;  
        #50 clk <= ~clk;  
        #50 clk <= ~clk;  
        // infinitely repeated  
    end  
endmodule
```

≡

```
module example ();  
    reg clk;  
    initial clk <= 0;  
    always begin  
        #50 clk <= ~clk;  
    end  
endmodule
```

Structured Procedures - always

- **always** can make simulator fall in infinite loop.
- Therefore, you have to limit the # of statements per a cycle.
- e.g., the former example executes a statement per 50 delay unit.

Assignment

- ALU implementation
 - A representative combinational circuit
- Due: Wed March 22 11:55 pm
 - -10% per day penalty for late submissions until Wed March 29 11:55 pm
 - 0 point after Wed March 29 11:55 pm
- Material
 - <http://www.asic-world.com/verilog/index.html>

Reference

- IEEE Standard for Verilog Hardware Description Language,
IEEE Computer Society